**Aim**

Investigate chaos in Lorenz equation using computational physics simulation and techniques using python.

**Summary**

The Lorenz Equations (above) provide a simple and yet significant area of study as it provides scientists a chaotic dynamical outlook of a system. We will be investigating different parameters of the Lorenz Equation to produce some of the most fascinating graphical outcomes of the trajectory of this 3 dimensional ODE. We will be applying what we learned in class using Runge-Kutta method to solve the non-analytical equations arising from its non-linearity.

**Introduction**

First derived by Edward Lorenz in 1963, the three-dimensional system came from a simplified model of convection system of the atmosphere. The trajectories of the system also converge into stable cycles which are now called strange attractors relating to fractals. These simple looking equations could have extreme dynamics over a wide range of parameters and producing chaos.

Where , r, b are parameters that we will be playing with.

is known as the Prandtl number.

r is known as Rayleigh number, dimensionless measure of difference in temperature of the top and bottom of the fluid.

b has no name, but also an important parameter. In convection problems, this is related to the aspect ratio of the rolls (Strogatz, 1994).

Chaos theory is defined as the qualitative study of unstable aperiodic behaviour in deterministic nonlinear dynamical systems (Kellert, 1993). Lorenz discovered these trajectories onto a complicated set now called the Strange Attractors.

Popularized by media, even non mathematical or physics people are enchanted by the beauty of the chaos in the Lorenz equation. These fascination might attribute to the appeal of the patterns they produce and hence inspiring people with creative ideas and philosophy. In this report, we will be investigating what causes these patterns by exploring multiple parameters of the Lorenz Equations and tuning gathering data for each change in the system. Furthermore, we will look into how these patterns are produced and what they mean.

**Code Organization**

Structure

The program was built in Python 3.7 using Spyder. Here are the imports

A screenshot of a cell phone

Description automatically generated

Fig. 1. Imports required to run the code.

The basic structure of the code is fairly simple, I have called the initial values to random float values between 0 to 1, STEP size (time is discretized) and initialised the GUI window. A Python class was used to coordinate all of the attributes of the instance self to make convenient when calling out methods and other attributes.

This is then followed by recursive functions to estimate the trajectory of the ODEs and outputting it into 3-Dimensional graph using matplotlib. These recurring functions serves to solving the ODE using the execution of the Runge-Kutta to the fourth order method which serves as a good estimate approximation of the trajectory in phase space.

This is done using the formula below to compute the next value from the previous value . We introduce a step size *h* and where n are positive integers.

This method of estimation has a local truncational error on the order of while the total accumulated error is on the order .

This method is known as the *Runge-Kutta in the fourth order*. The code for this is as follows:

A screenshot of a cell phone

Description automatically generated

Fig. 2. Main for loop to gather trajectory data, stored in self.res[i], where i = 0,1,2

The for loop creates an iteration of ﻿100000 and recursively calls function as shown in Fig. 3.

User Interface

The program returns a system report on the kernel as illustrated below.

A picture containing bird

Description automatically generated

Fig. 2. System Report printed in Kernel of Spyder.

An interactive python GUI app that allows user to take in different values of r, sigma and b using the module Tkinter. The 3D graph is embedded into the Tkinter frame so that the user can manually enter parameters of their choice.

A screenshot of a cell phone

Description automatically generated

Fig. 1. Main UI of Lorenz Graph Plotter.

Error checks are also enforced when user enters non-integer or non-float values.

To simplify the program and improve running time of the program, we are only simulating **one** particle undergoing the Lorenz trajectory. There is also no point in running multiple particle as we are only concerned with its trajectory after 100000 iterations.

A screenshot of a cell phone

Description automatically generated

Fig. 2. Graph is plotted when user clicks Run.

**Survey**

In forming the code, it was initially tough to get the simulations running. In order to have a neater way of the simulation, I decided to gather these function into a class, to call the data as self.res[i] where i = 0,1,2 for convenience in playing with the coordinates obtained.

One problem that arised is that, the canvas used to draw the data of the graph does not automatically clear when it is pressed “Run”. The method ﻿“FigureCanvasTkAgg” does not contain any “clear”, or “delete” attributes in its built-in method. Hence the simplest way to clear the data is to press “Clear” button on the app. This simply closes and reopens the app.

**Results**

To begin showing the results, we will fix two parameters sigma = 10 and b = 8/3 as they correspond to the critical Rayleigh number.

Properties of the Lorenz Equations

1. Non-linear: This is due to the quadratic terms xz and xy.
2. Symmetrical: Replacing , the Lorenz equations are still the same. i.e. if is the solution then so is .
3. Volume contraction: Volume in phase space shrink exponentially fast.

Let us breakdown the Lorenz equation into its two different fixed points. This is simply obtained by setting . Solving these equations will yield two fixed points that Lorenz called C+ and C-.

C+ ~ (

C- ~ (

Obtaining the stability properties of these fixed points by forming a Jacobian matrix of the linearized system. i.e.

Obtaining the eigenvalues by using the characteristic equation as

We notice that if , all eigenvalues are < 0. This means;

Hence, having a r < 1 implies that all eigenvalues are negative, and ultimately implies that the origin is a stable node. Now we plot using our program to check.

1. r = 0.1

A screenshot of a cell phone

Description automatically generated

Fig. 4. Output for r = 0.1

1. r = 0.5

A screenshot of a cell phone

Description automatically generated

Fig. 5. Output for r = 0.5

I did not Clear the previous data on the canvas so we can see the trajectory of the previous one in comparison to the new one. By observation, there is no chaos in the data we obtained, this proves that at r < 1; there is no chaos.

**Discussion**

**References**

Strogatz, H. S. (1994). Nonlinear Dynamics and chaos.

Kellert, H. S. (1993). In the Wake of Chaos: Unpredictable Order in Dynamical Systems. p.2

**Appendix**

﻿Code for the GUI app is below.

In case there are some errors in copy – pasting the code, get the original copy here: <https://github.com/sadfool1/Computational_Physics/blob/master/Final%20Project/Final%20Code.py>

"""

Author: James Morillo

Matric: U1740375H

SPMS

Computational Physics Final Project

Project title: "Lorenz Equations"

"""

#Initialise Imports

import sys

import traceback

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

from random import randint

import random

from tkinter import \*

import tkinter as tk

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg, NavigationToolbar2Tk #allows importing of a interactive graph

from matplotlib.figure import Figure

from matplotlib import style

from mpl\_toolkits.mplot3d import Axes3D #imports the 3D

import time

from tkinter import TclError

import matplotlib

matplotlib.use("TkAgg") #Backend of Matplotlib and we pull out TkAgg

class LorenzAttractorRungeKutta():

DT = 1e-3 # Differential interval

STEP = 100000 # Time step count (discretized)

X\_0, Y\_0, Z\_0 = random.random(),random.random(),random.random() #randomised initial values

def \_\_init\_\_(self,\*args, \*\*kwargs):

"""

====================

Initiliasing global

variables to be used

in other functions

====================

"""

global user\_r\_entry

global user\_sigma\_entry

global user\_b\_entry

global user\_r\_entry

global user\_sigma\_entry

global user\_b\_entry

global graphframe

global root

"""

=====================

Initialising the GUI

window, adding title

and initialising into

a grid for easy grid

management

=====================

"""

self.root = Tk()

self.root.grid()

self.root.title("Lorenz Simulation")

self.root.geometry = ("700x700")

self.root.resizable = (True, True)

self.res = [[], [], []] #initialise list to keep the values

controlframe = LabelFrame(self.root, text = "Parameter Control")

controlframe.grid(row = 1, column = 0) #this creates a frame for the entries

#DoubleVar allows the user to enter float values

user\_r\_entry = DoubleVar()

r\_entry = Entry(controlframe, textvariable = user\_r\_entry).grid(row = 1, column = 0)

self.root.r\_label = Label(controlframe, text="r value", height = 1, width = 12).grid(row=2, column=0, columnspan=1)

#DoubleVar allows the user to enter float values

user\_sigma\_entry = DoubleVar()

sigma\_entry = Entry(controlframe, textvariable = user\_sigma\_entry).grid(row = 1, column = 1)

self.root.sigma\_label = Label(controlframe, text="sigma value", height = 1, width = 12).grid(row=2, column=1, columnspan=1)

#DoubleVar allows the user to enter float values

user\_b\_entry = DoubleVar()

b\_entry = Entry(controlframe, textvariable = user\_b\_entry).grid(row = 1, column = 2)

self.root.b\_label = Label(controlframe, text="b value", height = 1, width = 12).grid(row=2, column=2, columnspan=1)

"""

============================

3 Buttons to Plot, Clear or

Quit

============================

"""

self.plot\_button1 = Button (self.root,

command = self.click1,

height = 2,

width = 8,

text = "Run").grid(row = 2, column = 0)

self.plot\_button2 = Button (self.root,

command = self.Quit,

height = 2,

width = 8,

text = "Quit").grid(row = 4, column = 0)

self.plot\_button3 = Button (self.root,

command = self.clear,

height = 2,

width = 8,

text = "Clear").grid(row = 3, column = 0)

graphframe = LabelFrame (self.root, text = "Graph") #creates a graph frame

graphframe.grid(row = 0, column = 0)

fig = Figure() #initialiase this into a Figure to be placed in canvas below

self.canvas = FigureCanvasTkAgg(fig, master = graphframe)

self.root.mainloop()

def click1(self):

"""

=================================================

This function is the main driver when the button

"Run" is clicked, where the main execution of the

Lorenz estimation using RK4 method.

=================================================

"""

global r\_info

global sigma\_info

global b\_info

global user\_r\_entry

global user\_sigma\_entry

global user\_b\_entry

global graphframe

try:

r\_info = user\_r\_entry.get() #This obtains the user input for r

sigma\_info = user\_sigma\_entry.get() #This obtains the user input for sigma

b\_info = user\_b\_entry.get() #This obtains the user input for b

print ("")

print ("==============================================")

print (" SYSTEM REPORT ")

timeinit = time.process\_time() #start timer to get execution time

print ("User entered: \n r = %f \n sigma = %f \n b = %f \n"

% (r\_info, sigma\_info, b\_info)) #Printing System reports in kernel

print ("The randomised initial values are \n X0 = %f \n Y0 = %f \n Z0 = %f \n "

%(self.X\_0, self.Y\_0, self.Z\_0))#Printing System reports in kernel

xyz = [self.X\_0, self.Y\_0, self.Z\_0] #initialises xyz in a list using the initial values

for \_ in range(self.STEP): #iterates up till the STEP size then applies RK4

k\_0 = self.\_\_lorenz(xyz)

k\_1 = self.\_\_lorenz([x + k \* self.DT / 2 for x, k in zip(xyz, k\_0)])

k\_2 = self.\_\_lorenz([x + k \* self.DT / 2 for x, k in zip(xyz, k\_1)])

k\_3 = self.\_\_lorenz([x + k \* self.DT for x, k in zip(xyz, k\_2)])

for i in range(3):

xyz[i] += (k\_0[i] + 2 \* k\_1[i] + 2 \* k\_2[i] + k\_3[i]) \

\* self.DT / 6.0

self.res[i].append(xyz[i])

fig = Figure()

ax = Axes3D(fig)

ax.set\_xlabel("x")

ax.set\_ylabel("y")

ax.set\_zlabel("z")

self.canvas = FigureCanvasTkAgg(fig, master = graphframe) #This embeds the graph into Tkinter, places this with its master in graphframe

ax.plot(self.res[0], self.res[1], self.res[2], color="blue", lw=1)

self.canvas.draw() #main app that draws and embeds the graph onto tkinter app

self.canvas.get\_tk\_widget().grid(row = 0, column = 0) #.grid places the object on the window.

timeend = time.process\_time() #timer for end of time

timer = timeend - timeinit #this obtians the time taken to excecute

print ("Time taken to execute: %f seconds " % timer)

print (" END OF REPORT ")

print ("==============================================")

except tk.TclError:

print (messagebox.showinfo("Invalid!", "Please Enter Valid Inputs (i.e. integers or floats)"))

print ("")

def clear(self): #FigureCanvasTkAgg has no module to delete canvas, hence i am forcing close and reopen of the app

self.root.destroy() #destroy main app

LorenzAttractorRungeKutta() #reset the application into original state.

def \_\_lorenz(self, xyz): #Lorenz equation returned in a list

global r\_info

global sigma\_info

global b\_info

#re-assigning the values to match the equation

p = sigma\_info

r = r\_info

b = b\_info

return [

-p \* xyz[0] + p \* xyz[1],

-xyz[0] \* xyz[2] + r \* xyz[0] - xyz[1],

xyz[0] \* xyz[1] - b \* xyz[2]

] #return as a list

def Quit(self):

print ("Exiting Program...")

self.root.quit() #Quit program

if \_\_name\_\_ == '\_\_main\_\_':

try:

LorenzAttractorRungeKutta() #run the class

except Exception as e:

traceback.print\_exc()

sys.exit(1)

(10 marks) final versions of your code in the Appendix;

(4 marks) report your final results;

(3 marks) discuss the physics of these results; and

(3 marks) list your references.